Design and implementation of an Android application to analyze the mobility patterns of citizens in Barcelona’s Metropolitan Region

MobilitApp

Author: Marc Llahona
Director: Mónica Aguilar Igartua
Codirector: Sílvia Puglisi
Date: July 14th, 2014
Acknowledgments

First of all I would like to thank Mónica and Jordi for giving the opportunity to work in this fantastic project, and the help they provide to me.

I would especially like to thank Silvia, who was always there to help me in all the problems and difficulties I had while working on this project, offering advice, counsel and content.

To my friends Carlos, Cesar, Roger and Marc, you always helped me to go through everything it happened to me in my life. No matter where we are in the future, or what we will be doing, our friendship will last forever. Gràcies amics meus.

To my family, my mom Rosa, my dad Francisco and my little brother Eric. You gave me a life, a home, you taught me almost everything I know, you supported and helped me in all the decisions I took in my life, and you loved me under all the circumstances. Us estimo molt família.

Last but not least, to the most important person in this world, the love of my life and my future wife Crystal. You always have been there, helping me, supporting me and encouraging me in the good and bad moments. Your love makes me strong and without you I would not be who I am today. T’estimo moltíssim princesa meva.
Abstract

In our project we have designed an Android application to obtain mobility data of the citizens in the metropolitan area of Barcelona. Our implementation synchronously obtains on the one hand, periodic location updates, and on the other hand, the type of activity citizens are doing. From these data are extracted mobility parameters of interest regarding citizens. All these data are processed together and saved in files, which are stored further extract them and obtain mobility patterns from citizens that could help improve the current transportation infrastructure.
Resum

En el nostre projecte hem dissenyat una aplicació Android amb l’objectiu de obtenir les dades de mobilitat dels ciutadans de l’àrea metropolitana de Barcelona. La nostra aplicació obté, per una banda, localitzacions periodiques del usuari, i per l’altra, el tipus d’activitat que est realitzant, de forma síncrona. D’aquestes dades se n’extreuen diversos paràmetres de mobilitat d’interès sobre l’usuari, i totes són processades de forma conjunta. Les dades queden emmagatzemades en arxius per tal d’extreure-’n posteriorment patrons de mobilitat que puguin ajudar a millorar les actuels infrastructures de transport.
Resumen

En nuestro proyecto hemos diseado una aplicacin Android con el objetivo de obtener los datos de movilidad de los ciudadanos del área metropolitana de Barcelona. Nuestra aplicacin obtiene, por un lado, localizaciones peridicas del usuario, y por otro, el tipo de actividad que ste est realizando, de forma sincronizada. De estos datos se extraen diversos parmetros de movilidad de interés sobre el usuario, y todos ellos son procesados de forma conjunta. Los datos quedan almacenados en archivos para extraer posteriormente patrones de movilidad que puedan ayudar a mejorar las actuales infraestructuras de transporte.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>ii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Resum</td>
<td>iv</td>
</tr>
<tr>
<td>Resumen</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>Glossary</td>
<td>x</td>
</tr>
</tbody>
</table>

## I Mobility Patterns of Citizens

1 Introduction

1.1 Barcelona Metropolitan Transport Authority
   1.1.1 Metropolitan Region of Barcelona

1.2 Mobility Master Plan

   1.2.1 Problems

   1.2.2 Solutions

## II Main Design Criteria for MobilitApp

1 Design Criteria

2 Application Fundamentals for MobilitApp

   2.1 Activities

   2.2 Services

      2.2.1 Base Service

      2.2.2 Intent Service

   2.3 Broadcast Receivers

   2.4 Selection of Application Components

3 Location Components in MobilitApp

   3.1 Location Sources in Android devices

   3.2 Android Location’s Tools

      3.2.1 Location APIs

      3.2.2 Location Client

      3.2.3 Location Request

      3.2.4 Location Class
# List of Figures

1.1 Barcelona Metropolitan Region division into 582 zones ........................................... 3

2.1 Android Activity Lifecycle Diagram ................................................................. 11
2.2 Android Base Service Lifecycle Diagram ......................................................... 14
2.3 Android Intent Service Lifecycle Diagram ......................................................... 15
2.4 Android Broadcast Receiver Lifecycle ............................................................... 16

3.1 Android Location Platform API ........................................................................ 21
3.2 Fused Location Provider ................................................................................. 22

1.1 Checking location sources diagram ................................................................. 29
1.2 MobilitApp homepage layout ........................................................................ 33
1.3 Trip draw on the map using `drawLocation()` method .................................... 35
1.4 MobilitApp Layout ......................................................................................... 38

5.1 Diagram of MobilitApp .................................................................................... 53

1.1 Screenshot of results obtained with MobilitApp ............................................. 55
1.2 Detectable Activity Types by MobilitApp ......................................................... 56
1.3 Screenshot of results obtained with MobilitApp ............................................. 57
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Comparison of battery drain per hour of Android location sources in 3 commercial devices</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Location Request priority modes</td>
<td>23</td>
</tr>
</tbody>
</table>
## Glossary of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>APK</td>
<td>Application Package</td>
</tr>
<tr>
<td>APP</td>
<td>Application</td>
</tr>
<tr>
<td>ATM</td>
<td>Metropolitan Transport Authority</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>MANET</td>
<td>Mobile Ad-hoc NETwork</td>
</tr>
<tr>
<td>UPC</td>
<td>Universitat Politècnica de Catalunya</td>
</tr>
<tr>
<td>VANET</td>
<td>Vehicular Ad-hoc NETwork</td>
</tr>
<tr>
<td>Wi-Fi</td>
<td>Wireless Fidelity</td>
</tr>
<tr>
<td>WPS</td>
<td>Wi-Fi-based Positioning System</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
Part I

Mobility Patterns of Citizens
1 Introduction

In this project, we are going to design and implement an Android application to obtain data regarding the mobility of citizens in the metropolitan region of Barcelona. In collaboration with ATM, this project’s main objective is to use these data to determine mobility patterns that could be used to improve current transportation infrastructure.

1.1 Barcelona Metropolitan Transport Authority

The ATM is an inter-administration partnership between administrations with transport service rights, created to coordinate public transport within the Barcelona Metropolitan Region:

- Government of Catalonia
- Local administrations
  - Barcelona Corporation
  - Metropolitan Transport Body (AMB, 36 municipalities)
  - Corporations with urban transport services

Its principal objective is planning the whole transport infrastructure in the metropolitan region of Barcelona. This process is called Mobility Master Plan, and it is done every four years and basically tries to address the following topics:

- Transportation and Infrastructure planning and execution.
- Coordination of services offered by (public and private) operators.
- Economics of the metropolitan transportation system.
  - Contracts with operators: regulation and competition.
  - Fare policy: fares integration and the annual price review.
  - Funding and financing.
- Communication: definition and promotion of the system’s corporate image.

1.1.1 Metropolitan Region of Barcelona

When designing our application, we will use as input all features, scenarios and possible circumstances that may occur in the area where we want to track citizens mobility. In our case, our application will be tested and used in Barcelona’s metropolitan region, which has a total surface of 3,240 Km$^2$ with more than 5.0 million of citizens divided. Only the city of Barcelona has 100 Km$^2$ of the surface and a population of 1.6 million people. For planning purposes, ATM has divided the region into 582 zones, of which 203 are in the city of Barcelona.
1.2 Mobility Master Plan

ATM’s Mobility Master plan is based on data obtained from citizens either by understanding how they travel through the tickets sold and used, or by directly surveying them. However, these data are actually very poor to build an accurate Mobility Master Plan. Problems like lack of reliability or getting only partial mobility samples are just some of the problems that we want to solve providing better mobility data.

1.2.1 Problems

Current ATM Data Recollection Methods

Basically, ATM has two different methods to recollect mobility data from citizens:

1. Monitoring a bit included in only 10% of the metropolitan region’s fare integration tickets. ATM is capable of reconstructing the different stages of each trip when these are validated in transport stations. The main idea of this estimation process is the assumption that every time user validates the ticket in origin station, this station is at the same time considered the destination station of the previous trip.

2. Telephonic home surveys to the citizens.

Data Challenges

The data obtained by ATM are not always reliable. Telephonic home surveys data do not have any kind of reliability more than trusting citizens are telling the truth. But not only that, information provided is based on memory and interpretation of citizens, which means data are far away from being reliable based on empirical results. Reconstruction of trip stages is also based on interpretation, because it is assumed that all original stations are, at the same time, destination stations in previous trips, which are an assumption that could be false. In fact, no real information of destination stations is obtained, which means the estimation is done with biased
information. At the same time, both methods cannot be representative of the entire metropolitan area of Barcelona, because the percentage of population used to obtain such data is very low.

1.2.2 Solutions

Smart Citizens as Sensors

The first thing is turn the citizens of metropolitan region of Barcelona in smart citizens, acting as sensors themselves through their smartphone devices, contributing to create mobility data. This will increase the reliability because data will be directly obtained from citizens without any interpretation as well as it will increase the validity of data because we can be able to obtain them from a broader spectrum of the population. This will result in better mobility patterns, that will improve ATM’s Mobility Master Plan, and at the end will improve transportation services in the region.

Android Platform

Since we are going to need to use citizens’ smartphone devices in order to obtain mobility data, we have to choose the mobile platform that can meet our needs. This platform is Android OS, because it can provide us with the following features:

- It is an open source based on Linux and written in C, C++ and Java, all of it free to use for development.
- Built-in Services: Android comes with multiple ready-to-use services and features with infinite possibilities.
- Large audience: It is the most used platform in the world, with more than 80% of smart devices markets running some version of Android.
- Location sources: Android devices typically come with the latest technological equipment and techniques to obtain location samples in the most efficient and accurate possible way.

MobilitApp

In this project, we are going to create a new Android application called MobilitApp that aims to achieve an implementation with capability to detect citizens daily trip, identifying location, activity type, distance, duration and average speed, when they move within the metropolitan region of Barcelona. The next chapters we are going to explain what criteria were established to design the application, the architecture of its implementation, and finally we will provide the principal results obtained from multiple tests.

---

1 Currently there are multiple projects under development UPC that have to do with smart citizens.
Part II

Main Design Criteria for MobilitApp
1 Design Criteria

The main objective of MobilitApp application is to provide a solution to obtain more reliable and representative data of the mobility of citizens in Barcelona’s Metropolitan Region. That can lead to better results on the estimation of mobility patterns. In our application, we are going to design a method that will get and process periodic samples of location and activity of citizens, by means of existing sources in Android devices. This will provide complete tracking of the citizens movement, along with other useful information related like distance, duration, or speed, that will be written to files, and stored for further developer analysis. The application includes the ability to read such files to show citizens their daily activity and information on a map.

In our MobilitApp application we have determined the most important criteria that we have to follow when designing it. These criteria will help us to identify the aspects and functionalities that are most critical for the application, and that will have a relevant impact to its subsequent implementation, like how structure it, how users will interact with it, which components use and how to use them, Android services and classes that are best suitable for the type of data we want to handle, or how to structure processed data and how store them. The main principles determined are the following:

Processing components running continuously in background

The most important functionality of MobilitApp is to receive periodic updates regarding location and detected activity of the users, and then process these collected samples in order to obtain the mobility data. On this basis, the components responsible for these tasks must keep running independently of the current state of the application, either foreground, visible or background, or if the user is actually interacting with MobilitApp or not. Therefore, these components must be designed to continuously run in the background of Android OS, trying to have the minimum impact possible over the system performance that could disrupt user normal activity with the Android OS, and other applications or services. This is, undoubtedly, the fundamental criterion for a successful design and implementation of MobilitApp. The rest of the criteria is directly based on this one.

Low power consumption

As the core components of the application must continuously run in the background, device power consumption may become a critical issue, especially if it is not properly addressed. Therefore, the design of MobilitApp must be oriented to achieve a low power consumption solution allowing users to use our application without experiencing a significant battery drain problem that might disrupt their usual activity with Android device. Our goal is to achieve an average power usage below 2% per hour, which would allow the battery last at least one day, under non-intensive usage conditions. This factor will condition the choice of existing sources and services in Android devices to get location and detect activity, the design of functions to obtain periodic samples, and the design of processing algorithms of data.

Requirements for mobility data

In our application we need to identify the most relevant parameters that characterize mobility data, in our case the location and the activity recognition, and establish the adequate thresholds for their acquirement and processing in order to ensure that these data have the minimum

---

1 We consider intensive usage conditions those in which the user plays several game applications, listens to music (either online or offline), or plays videos during the day without charging the battery or setting power saving options on the device.
conditions for our objectives, i.e., to deduce mobility patterns of citizens. These parameters include accuracy, confidence level, coverage, minimum update intervals, and minimum segmentation length for mobility data.

- **Structure of mobility data:** Data organized as daily trips and broken down to smaller parts called segments, that include location samples, estimated activity during the segment, and distance, duration, and average speed consumed (if any). That requires a synchronization between the location samples and activity recognition samples in order the build segment data structure. Each daily trip is therefore built from multiple segments.

- **"Block" level accuracy for location:** In MobilitApp, we want to track behavior of citizens when moving between places and areas of metropolitan region during long periods of time, which means that accuracy is actually not a critical parameter to meet, because areas and places are large enough to allow us certain margin of error that would not significantly affect the quality of mobility data. Therefore, "block" level accuracy for positioning will be sufficient. Block level accuracy considers that the difference between location estimated and the real location of citizen will never be higher than 150 meters, which means that any location sample obtained in our application will have a precision range from the minimum possible (depends on location sources used by the Android device, but typically around 10 meters) to 150 meters. With this accuracy range, we will also help to keep the average power consumption of the application below desired threshold of 2%. Accuracy and power consumption has a direct relationship, a better accuracy typically requires higher power usage. As we said, a positioning range more accurate not only would not give a significantly improvement of quality, but also would incur in higher power consumption. Instead, losing this margin allows us to reduce, at the same time, the average power consumption in a way that will improve MobilitApp’s performance, which is most important in this context.

- **Maximum coverage for location:** Continuous tracking of citizens requires getting their position in multiple scenarios while keeping the accuracy and average power consumption requirements. This means that our application must be able to keep getting position updates when citizens are outside (for example freeways, roads, streets, parks, beaches, hills, or open fields) and when they are inside (buildings, malls, shops, within transports, or even underground) with a block level precision, as long as some wireless signals are available. Therefore, we must select those location sources that can give us the maximum possible coverage for all places where citizens may be within the metropolitan region. Be because we have determined a block level accuracy range we will be in a position to monitor citizens’ mobility in almost any scenario without losing noteworthy quality in our data. In contrast, we could expect a small increase of average power consumption in scenarios with very low or non-existent wireless signals, in counterpart by such coverage.

- **High confidence level of detected activities:** Our application requires the maximum feasible level of confidence when recognizing the activity of the citizens. Specifically, we expect to reach at least 75% of confidence that we are estimating the right activity, which includes the recognition of activity itself, as well as the activity and locations estimation as a single segment. To facilitate the estimation, we will establish a set of predetermined general activities that citizens may do when moving in the Metropolitan Region. Therefore, we have to implement a function that will allow us to obtain such desired level of confidence while synchronizing it with the location samples updates.

- **Time intervals for updates:** The last relevant parameter to obtain mobility data is the frequency interval for updates of location, activity and segment samples. Each interval time has been decided on different purposes that are pertinent to each one of the types of data.

---

2 We are not capable to detect where exactly the citizens are, but it is something that is under development. For more information see Future Research & Development part.
- **Segments** are fragments of citizens’ daily trips and therefore this type of data has to establish which is the minimum time that we do consider that an activity has to be detected and processed. On the one side, if we take very short fragments the value of the samples will decrease because the margin error of each one of them may significantly increase, becoming in useless data for mobility pattern analysis purposes. On the other side, if we take very long fragments we may be unable to detect some useful minor activities that could be interesting from the point of view of mobility patterns in city neighborhoods or in smaller towns in the metropolitan region. When implementing *MobilitApp*, we will consider the smallest segment time, that will provide the most useful data and results for our purposes, it will be 120 to 150 seconds (2 - 2.5 minutes).

- **Location samples** must be updated in intervals that will allow us to track their moves within the metropolitan region in order to detect mobility patterns. The key to determining this time is to balance the tracking precision and the average power consumption. On the one side, if we take a very long time span between location updates, we will have a minimum average power consumption but at the same time the value and quality of data will certainly decrease. On the other side, if we take a very short time span, we will have the opposite case, an average power consumption much higher than our threshold but will get very accurate mobility data. Based on Google’s location guidelines[6], we have decided that the time interval updates between location samples, which we will implement in *MobilitApp*, will be 20 seconds.

- **Activity recognition samples** must be updated in intervals that will allow us to estimate, with a confidence level around 75%, citizens’ activity when they moving within the metropolitan region. To achieve such confidence, we will need multiple samples in very short periods of time in order to not miss or to not do wrong estimation, of citizens activities that can provide useful pattern data. Using a long time span would reduce the confidence level and the value of the activity samples. In the same direction, time interval cannot be longer than time interval of location updates, in order to ensure adequate synchronization of both types of data and merge them into a segment data format.

**Non-intensive interaction of users with application**

Our concept of how users will interact with the *MobilitApp* application is based on the most important criterion that says the core components of the application will run in background. Our idea is that the users will not use very often (no more than a couple of times per day in normal conditions) or during long periods of time (no longer than 5 to 10 minutes). For that reason, in order to stimulate citizens to use the application, on one side, we will design a simple, intuitive and attractive layout that will provide a great visual effect on the users, and on the other side, we will add functions that users will use to see and get their own mobility data, that will include functions like current location, location history of previous days, or information (distance, duration, and average speed) of their trips, overall and per segment.

**Data storage**

Our application requires to store mobility data in often periodically intervals because we want users being able to see their current activity. We also want them to see past daily activities. Therefore we have to choose a data format the is fast to write, read and process for further analysis.
2 Application Fundamentals for MobilitApp

In this chapter, we are going to analyze with greater depth existing application components, and its associated processes, in Android platform. We will review on their roles, their most important functionalities, and their lifecycles in order to know exactly how they work and to determine which components will be included in our design guidelines to implement our MobilitApp application. The selection of the application components and will be made following one of our main design criteria that establishes that they must be continuously running in background.

Application Components

A component[7] is a unique entity with a specific role that is essential to run any application. Every Android application is made up of a combination of up to fours different types of components (Activity, Service, Broadcast Receiver and Content Provider). There is no requirement stating that all types must be included, and neither there is a limitation of how many components of each type can be implemented in the same application.

Processes and Threads

A process is an entity that runs the application components while a thread is the element in charge of dispatching events to the appropriate UI widgets[8]. Applications are by default sharing a process for all its components (activities, services, and receivers) and running on the main or UI thread, unless otherwise specifically implemented.

In Android platform there are four main different types of processes depending on their level of importance:

- **Foreground**: A process that is required to directly interact with the user. Foreground processes can run the following components:
  - An activity is running a foreground process when the user is interacting with it.
  - A service is running a foreground process when it is either bound to an activity that the user is interacting with, explicitly running in the foreground mode, or executing one of its lifecycle callbacks.
  - A broadcast receiver is running a foreground process when it is executing its `onReceive()` method.

- **Visible**: A process that has lost focus on behalf on another process that is foreground but it still can affect interaction of user with application. Visible processes can run the following components:
  - An activity is running a visible process when it is not in the foreground, but is still visible to the user.
  - A service is running a visible process when is bound to a visible or foreground activity.

- **Service**: A process that is running a service that has been started with the `startService()` method and does not fall into either of the two higher categories. This process has lower priority than foreground and visible processes, but higher compared to background, and therefore it is the ideal type process to perform long operations in the background of Android system.
• **Background:** A process that is running an activity (and associated components) that is not visible to the user. These processes have lower priority than foreground, visible, or service process, and therefore will be the first ones to be killed by the system if it is running low on memory.

### 2.1 Activities

**Description**

Activity is a component that provides a single screen that users can interact to do something. To create an activity, first it must be created a subclass of it, also called activity. The implementation of the GUI and all its elements, like text fields, lists, tabs, buttons, widgets, and almost any kind view, must be done in this subclass. Rest of application components cannot implement these functionalities. Activities can also implement fragments. These are pieces of an application’s user interface or behavior that can be placed in an activity. Fragments are good to implement, for example, a map of the activity. Typically, applications have at least one initial activity, called **Main Activity**, that is launched when the application is launched, and it is entry point to call or create other components. Therefore, activity’s major role is interact with the users providing them a layout and functions, either interactive or not, to use the application.

**Lifecycle**

However, the most critical part of every application component understand how it work through its lifecycle. It is important in order to identify all possible states of an activity to properly implement our functions, especially in the Main Activity, and ensures that our application will work in the way we wanted in order to meet our design criteria. In the activity lifecycle, whose diagram is shown in figure 2.1, we have different aspects to focus on.

First of all, we need to identify which are the states of activity. Activity transition between states according to a predetermined logic depending on the type of interaction that the user is having with our activity. Each state has specific features and are attached to a different type of process. An activity has three major states:

1. **Active State.** The activity is running on the foreground of the screen in the Android device. In this state, the users will be seeing and interacting with the activity. The activity is automatically placed at the top of the activity stack. This stack is the place where all the activities are arranged according to their level of importance, that depends on the level of importance of the underlying process that is running the component operations. An activity that is on top of the stack is running with foreground process running that has the maximum level of importance. This means that an activity in an active state will be the last process to be killed by the Android system under low memory situations. Actually, the only situation where an activity in an active state with a foreground process can be killed is when the system is so low in memory that cannot continue to run.

2. **Paused State.** The activity is running below other foreground components. In this state, the activity is still visible to the user although has lost focus, remains completely functional, maintains all memory states, and keep working, otherwise determined when paused. The activity is no longer on the top of the stack and the process that is running its operations has now decreased its level of importance, from foreground to visible. Therefore, the activity will be killed before, regardless of the fact that this will only happen in extremely low memory situations. From this state, besides being killed by the system, the activity can be resumed by the user and brought back to active state, or can be transitioned to the stopped state.
Figure 2.1: Android Activity Lifecycle Diagram
Stopped State. The activity is running in the background below the foreground and visible activities, and maybe additional application components. In this state, the activity is no longer visible by the user, ceases to be fully functional, but may still be doing some work unless determined when stopped and still maintains all memory states. The process running the activity is now labeled as background, which means that in situations of low memory this activity will be one of the first to be killed. From this state, besides being killed by the system, the activity can be restarted by the user and brought back to the activity, or it can be destroyed.

The second aspect that we have to analyze are the working loops that monitor the most important transitions within an activity. There are three major working loops:

- **Activity loop:** This loop covers the whole activity lifetime, from its creation to its destruction. To create an activity, `onCreate()` method must be called. This method is where all global setup of activity must be implemented, such defining layout or views. Activity cannot be killed while implementing `onCreate()` commands. To destroy the activity, `onDestroy()` method must be called. Activity can be destroyed either because the activity ended its tasks and finished itself, the user finished it, or the Android system finished due low memory situations. It is important to notice that both methods can only be called once during the activity lifetime, which also means that any function implemented within them cannot be use multiple times, unless that activity is recreated.

- **Visible loop:** This loop covers what happens in the activity after its creation but before its destruction. It starts when `onStart()` method is called, and ends when `onStop()` is called. Between this methods the activity visible and fully functional but without being in the foreground to interact with the user. Both methods can be called multiple times as the activity becomes visible or hidden to the user. Every time the activity calls `onStart()` method from a stopped state, Activity may also implement a method called `onRestart()`, in case some functions or states must be reimplemented before start activity again. Activity cannot be killed while implementing `onStart()` or `onRestart()` commands, but it may be when is in stop state. This worker loop is typically used to monitor changes during the activity lifetime of the activity. However, it is less important and useful than the other two.

- **Foreground loop:** This loop covers what happens in the activity between the active state and the paused state. It starts when `onResume()` method is called transitioning the activity process into foreground, and ends when `onPause()` is called, transitioning the process from foreground into visible. Both methods can be called multiple times as the user interact with the activity or not. Actually, this transitioning through this loop is very frequent in activities, and typically used to implement functions that has to be called several times during activity lifetime, like deliver results or start new intents. This is the most important loop in activity lifetime.

With all this information, we can conclude that activities are designed and intended to work in foreground or when they are still visible to the users, but not in background because their main roles is implement GUI layouts. Therefore, long running task should not be implemented using activity components.

### 2.2 Services

Description

Service is a non-GUI component that runs in the background, usually to perform long-running tasks that will not interact with the users. Services do not create their own thread to run, instead they run on the main thread of the application. Asynchronous or new threads can be
implemented within them in order to perform intensive or blocking tasks. Services run also in the
same process of the application that called them, unless otherwise specified creating and running
its own independent process (foreground or service). In general, This means that services have
the same level of importance of the activity process (foreground, visible or background), and
therefore share their fate. If the activity process is killed, the service cannot keep working, and
if it is not properly stopped or destroyed, it may be keep running as a zombie component in the
system. To create a service, first it must be created a subclass of it, also called service, or one of
its subclasses.

Basically, there are two main type of services depending on the task to perform that could be
used in our application: base service and intent service (a subclass of the base service class).

2.2.1 Base Service

Description

Base service is the generic class of service[12]. It is suitable for almost any task that a service
can perform. By default it runs on the main thread of the application and shares the same
process. If the service must perform intensive long-running tasks, a new thread can be created
to prevent slow down the main thread, where other applications are running in the system. Base
services can also run within a new process in those situations where the service may require
different importance level than the activity, like for example a user that do not interact with the
application but it uses intensively its services in the background.

Lifecycle

Base service’s lifecycle is much simpler than activity’s lifecycle, with only three important methods
and a single state. To start a base service we have to call startService() method. If the service has
not been created yet, the service calls onCreate() method, that will create it and it will implement
one-time setup functions, if any.

If the service has been already created, it will call then onStartCommand() method, instead of
calling the onCreate() method again, to put the service into active state. In active state, service
will run indefinitely in background until it is stop, either by the service itself when its task is
finished or by the user to stop it. As any component in Android platform, base service can be
destroyed by the system in low memory situations. As we said, base services share the processes
with application, except when a new process is started for the service itself.

In the first case, the service will be dependent on application process level of importance,
that at the same time will depend on user’s interaction with it. If the application process is
killed, the application will be destroyed, and with it the service. In the second case, the service
will be independent of the application, and therefore will be able to keep running even if the
application process is killed. This process can be either a foreground process or a service process,
which is determined when it is started. A service running with a foreground process will be less
likely to be killed and will run longer than any other service or activity, but as a counterpart it
will consume more system resources. It is suitable for services that do not require user’s direct
interaction, like music playback. Service process is specific for services to work in background,
but with a level of importance higher than regular background processes. This will help those to
keep running longer time if the system is low on memory, because they will be killed after the
background processes, but with a lesser resource consumption than a foreground service.

If the service is stopped, this actually means that its process will be killed, and therefore the
service destroyed. Another call from other component to start the service after being stopped,
in fact implies to recreate the service by calling onCreate() method, as a new service. Unlike
activities, services cannot save memory states to be recovered when they are recreated.
Figure 2.2: Android Base Service Lifecycle Diagram
2.2.2 Intent Service

Description

Intent service is a subclass of base service. Unlike base service, the intent service creates by default its own worker thread independent of the main thread of the application. This makes it suitable to perform short asynchronous and recurrent tasks that could overload application’s main thread.

Lifecycle

To start an intent service we can use two different calls depending in the type of task we want to perform. If we call \texttt{startService()} method we are going to create a service that will run one-time only, whereas if we call \texttt{getService()} method we are going to create a service that will be called recurrently to perform short tasks belonging to a longer operation implemented in other component.

![Android Intent Service Lifecycle Diagram](image)

Figure 2.3: Android Intent Service Lifecycle Diagram

After being called, the service is created and executed calling the methods \texttt{onCreate()} and
onStartCommand(), as happens with base services. After that onHandleIntent() method is called to process the requested tasks from the service. If created to work as a single asynchronous operation, the intent service will handle and implement multiple concurrent intents in a one-time run. This behavior is similar to a server responding multiple client requests. If the service was created as a recurrent operation, the intent service will handle only one intent at a time. This behavior is optimal to request periodic updates of data. In both cases, when all requests have been handled the intent service stop and destroy itself.

2.3 Broadcast Receivers

Description

According to [14], a broadcast receiver is a component that allows to register for system or application events. All registered receivers for an event are notified by the Android runtime once this event happens.

Lifecycle

Broadcast receiver lifecycle is the simplest one, with only one method and one state. When an intent is sent by using sendBroadcast() method, the receiver is created and calls its only method, onReceive(). While operating in this method, the receiver is in active state and running a foreground process. After the task is done, the receiver is destroyed. This component is very useful to perform synchronous operations.

![Android Broadcast Receiver Lifecycle](image)

Figure 2.4: Android Broadcast Receiver Lifecycle
2.4 Selection of Application Components

In our application will have only one activity, the Main Activity. The Main Activity is not suitable to implement long-running task to receive periodic location and activity recognition updates. It will be used to set up layout, user’s functions, a map, as well as to launch the services that will indeed perform location and activity detection operations. Therefore the loops that, in our case we will take more care of will be the activity, because we have to set up all the components in on create (we have to call services only once during activity lifetime) and the foreground loop, that will be active during user’s interaction with our application. We are not considering to perform any specific operation within the visible loop, as we do not have to monitor activity parameters.

Along with the Main Activity component, we are going to create two services. One service will obtain periodic activity updates while the other will obtain periodic location updates at the intervals decided in the design criteria. The most complex element in our application will be the synchronization of both services. We need to assure that activity samples actually correspond with location samples. Based on that, we have taken two decisions. First decision is that activity recognition service will be an intent service and location service will be a location service. The second decision is to use a broadcast receiver to synchronize both services. The idea is periodically send activity type samples to be received in the location service. Then location service, as a base service, will process both types of data together to obtain the best quality in mobility data to meet our design requirements.
3 Location Components in MobilitApp

3.1 Location Sources in Android devices

The selection of location sources is directly related to meet, on the one side, the low power consumption criterion, and on the other side, the specified requirements for mobility data. There are at least four location sources available in almost every Android device: GPS, WPS, Cell ID, and sensors. Each one of these sources has its own features regarding average power consumption, accuracy and coverage which basically depends on how location data is obtained as well as the minimum optimal update intervals to achieve with an acceptable level in quality of data.

Unfortunately, Android platform is not well documented regarding detailed profiles of average power consumption of each one of the location sources. For that reason, we are going to use approximations of their power usage profiles in order to select those that could meet our low power consumption criterion. To compare all the location sources we are going to consider the average power consumption relative to hypothetical percentage of battery power consumed by our application, if we were using only the location source under analysis to obtain updates of mobility data. Typically, in their specifications, manufacturers of Android devices provide the total amount of electrical current that battery can supply per hour, in mAh, instead of power capacity per hour.

Global Positioning System

GPS consists of up to 24 or more satellites broadcasting radio signals providing their locations, status and timestamp. The GPS receiver in the Android device can calculate the time difference between broadcast time and the time radio signal is received. When the device knows its distance from at least four different satellite signals, it can calculate its geographical position[15].

- **Average Power consumption**: GPS receiver works as an independent-powered component in the Android platform with the unique objective to obtain location samples. Therefore, all the battery power consumed by GPS is consequence of a user positioning operation. According to several references, we can estimate that the average power consumption of an active GPS with a location update interval between 5 and 15 seconds is approximately 125 to 145 mA.

- **Coverage**: GPS location coverage is probably the most limited of all four available sources. Although it is a source that cover the whole globe, it only provides good performance in outdoor scenarios, because it requires satellites’ visibility. GPS does not work in indoor locations (may provide some location data in very few of them if it has some kind of visibility of satellites, but with very poor performance) like buildings or undergrounds. Even in outdoor environments, GPS may suffer from poor performance, especially in cities, because objects like buildings, trees and other obstacles can overshadow visible satellites, decreasing its performance. Therefore, GPS range of coverage can be quite limited in a metropolitan region, compared to other sources, specifically if we want a continuous position tracking of the citizens.

- **Accuracy**: In outdoor scenarios with good satellite visibility, GPS is the positioning system that can provide the most accurate location of all four existing sources. In perfect conditions GPS can achieve a precision higher than 3 meters, but in optimal conditions the range is between 3 and 10 meters. In non-optimal conditions, i.e. lower visibility on the satellites, GPS accuracy range is typically between 30 and 100 meters. GPS is also capable to provide an estimation of altitude and speed along with location.
Wi-Fi-based Positioning System

WPS sends a location request to Google location server with a list of MAC that are currently visible by the device, then the server compares this list with a list of known MAC addresses of the device itself, and identifies associated geocoded locations. After that, Google server uses these locations to triangulate the approximate location of the user[16].

- **Average Power consumption:** Wi-Fi receiver works as an independent-powered component in the Android platform but multiple-purpose that is not restricted only to obtain location samples, as happens with GPS receiver. Average power consumption of WPS represents only a small part of power usage of Wi-Fi services. Because of this, we do not have to consider the power consumption of Wi-Fi when selecting location sources to meet our design criteria, but only the part that is actually consumed by obtaining location updates. According to multiple references, we are going to estimate that the average power consumption of an active WPS with a location update interval around 20 seconds is approximately 25 mA.

- **Coverage:** WPS location coverage is less broad than GPS but is more versatile. Its coverage is not global but block level, because it has to be in the range of a wireless signal to be able to provide location updates. However, within this range, it can provide good performance both outdoor and indoor scenarios. Therefore, WPS range of coverage can be very useful when tracking continuous locations within cities and towns.

- **Accuracy:** In indoor scenarios, WPS is the positioning system that can provide the most accurate location of all four existing sources. In normal conditions, WPS can achieve a precision between 3 and 10 meters. In outdoors, the accuracy of GPS decreases significantly being able to provide a range between 25 and 150 meters, depending on the external conditions of the environment and the signal strength.

Cell-ID based localization

Cell-ID based localization uses both the Location Area Code and Cell-ID that the Base Transceiver Station broadcasts. Android devices are always receiving these broadcast messages, which means that they always know their Cell-ID at any time. Knowing this, it can obtain an approximation of its actual location using the geographical coordinates of the corresponding Base Station[17].

- **Average Power consumption:** Since Android devices always know their Cell-ID, the average power consumption consequence of this positioning system is extremely low.

- **Coverage:** Cell-ID location coverage is global, as happens with GPS, and versatile, as happens with WPS, which implies that has the best coverage of all four existing location sources. Therefore, cell-ID has an adequate coverage range to continuously track location of citizens in the metropolitan region.

- **Accuracy:** Cell-ID based location system accuracy depends on cells size in the network that are connected with user’s Android device. This means that in those regions with smaller cells (typically in cities and metropolitan regions), cell-ID location will provide good performance achieving an accuracy range between 50 and 200 meters. Moreover, in regions with bigger cells (typically in rural areas) performance will significantly decrease, and cell-ID will only achieve an accuracy range between 1000 meters and several kilometers.

Sensors

- **Average Power consumption:** Android devices have usually several built-in sensors. However, not all of them can be used for positioning processes, being only accelerometer,
magnetic field, orientation, and gyroscope that one that can be used for such purpose. Based on their specifications, the power consumption of these sensors are the following:

- Accelerometer - 0.23 mA
- Magnetic Field - 6.8 mA
- Orientation - 13.13 mA
- Gyroscope - 6.1 mA

If all sensors were used together at the same time to obtain locations updates, we would have an average power consumption of 26.26 mA.

- **Coverage:** Sensors have a wide coverage because they can process multiple different types of data. From acceleration, that covers an individual range, to magnetic field, that covers a global range, sensors can provide data to improve location samples in almost any scenario.

- **Accuracy:** It depends on the type of sensor and how is used, which means it is infeasible to provide real accuracy values. However, according to Google, sensors improve the accuracy obtained with just simple WPS and Cell-ID.

**Summary**

In order to understand what suppose for an Android device the power consumption of each one of the four location sources, we are going to calculate the percentage of battery drain per hour on 3 existing commercial. To calculate this we are going to obtain the relation between the power consumption of the location source and the capacity of the battery of the Android device, multiplied by one hundred to get the percentage. The results shown in table 3.1 demonstrate that GPS average power consumption is too high for our low power consumption threshold of 2% from our design criteria whereas the other locations sources can be suitable for our application, at least in terms of power usage.

<table>
<thead>
<tr>
<th>% of battery drained per hour</th>
<th>GPS</th>
<th>WPS</th>
<th>Cell ID</th>
<th>Sensors (Combination)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GALAXY S2 (1650 mAh)</td>
<td>7.5 - 8.8</td>
<td>1.5</td>
<td>&lt;0.5</td>
<td>1.6</td>
</tr>
<tr>
<td>NEXUS (1750 mAh)</td>
<td>7.1 - 8.2</td>
<td>1.4</td>
<td>&lt;0.5</td>
<td>1.5</td>
</tr>
<tr>
<td>GALAXY S3 (2100 mAh)</td>
<td>5.9 - 6.9</td>
<td>1.2</td>
<td>&lt;0.5</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of battery drain per hour of Android location sources in 3 commercial devices

Summarizing, GPS location source will be discarded from our application because does not suit our design criteria of low average power consumption nor the range of coverage. It is true that it can provide the best outdoor accuracy, but this is not a priority in our implementation, because we establish a block level range that can be met by the other location sources.

The rest of location sources do actually meet our design criteria, but each one of them has strong features where to other are more weak. Therefore, instead of picking just one single source, we are going to obtain location data updates with an hybrid combination of WPS, cell-ID and sensors to obtain the quality required while meeting each one of the principles we set.

---

1. Information obtained from sensors available in a Samsung Galaxy S2 device
3.2 Android Location’s Tools

Android platform provide developers with multiple tools, including services, libraries, classes and methods to implement and develop their application. In this section we are going to review the most important location tools that Android platform provides, and select those ones that we are going to implement in our *MobilitApp* application.

3.2.1 Location APIs

In Android platform there are two APIs that can manage location sources along with location libraries, classes and methods to provide the best location data according to established requirements. These APIs are Android Location Platform and Google Fused Location Provider. We are going to take a deeper look in order to select determine its pros and cons to select the more adequate API to meet our design criteria as well as the one that can better implementation tools for our application.

**Android Location Platform**

Location platform[^19] is the generic API in Android to get location data. This API considers each hardware components (location sources) as independent. On one side, it provides the Sensor Manager class to handle location data that comes from all sensors, and on the other side provides the Location Manager class that handle data from GPS, WPS, and cell-ID. GPS data is provided by GPS location providers whereas data from WPS and cell-ID are wrapped together and they are provided by network location providers.

![Location Manager](image1.png)

*Figure 3.1: Android Location Platform API*

Location Manager class allows full control of location sources and providers in order to select the best location sample that meet established criteria. However, for a continuous tracking of citizens locations through multiple coverage areas and situations can turn its implementation into a really complex task, specially when we are dealing with strong power consumption requirements. Furthermore, if we want to also use the location data provided by sensors, that will add even more complexity to the application design.

**Google Fused Location Provider**

Fused Location Provider[^20] is an API created by Google to get location data. To use it, we need to install and use the Goole Play Services APK in our Android device because without it Fused Location Provider will not work. This API considers all hardware components as single location
sources layer that can be used according to predetermined criteria implemented on it. The same happens with the classes and providers that handle location data, they have been put together in the same layer and are managed with Location Client class. In that way we can use the location sources together without having to use different classes or providers, but with the capacity to select which ones we want to use based on power consumption profiles.

![Fused Location Provider](image)

Figure 3.2: Fused Location Provider

Location Client is more efficient using location tools, more efficient with average power consumption, and less complex to implement than Location Manager. Therefore, Fused Location Provider API along with all its classes and methods are the location tools we are going to use to implement location functionality our *MobilitApp* application.

### 3.2.2 Location Client

This is the main class from Fused Location Provider API to get location data[21]. Its principal functions are connect and disconnect to Google Location Services in background, and request location updates by starting a service, with the possibility of being a base service or an intent service.

### 3.2.3 Location Request

This is the class[22] that sets up the main parameters to meet our design criteria for location requests from the Location Client class. The most relevant parameters are the following ones:

- **Desired interval time**: This parameter establishes the ideal interval time between location updates obtained by the Location Client. However, it is not an exact interval, which means that locations can be obtained in lower or higher interval times, depending on external conditions. According to our design criteria we should set this parameter to 20 seconds.

- **Fastest interval time**: This parameter establishes the minimum interval time between location updates that can be obtained by the Location Client. Unlike desired interval time, this is an exact interval, which means that location updates will not be obtained in lower intervals of time than set by this parameter. According to our design criteria we should set this parameter to 20 seconds.

- **Smallest displacement**: This parameter establishes the minimum distance that must exist between two location updates in order to consider that the user’s location has actually
changed and that the new sample must be obtained by the Location Client. This parameter can be useful to determine when we consider that a citizen is actually moving between places and no just moving around the same place, which has not value as mobility data.

- **Priority mode:** This parameter establishes the priority of the location requests based on the location sources that we want to use and the accuracy of location samples we require. There are the following modes:

  - **High Accuracy:** This is the highest priority mode. Using this mode Location Client will obtain the best location available using all the location sources available together. This is also the mode with higher average power consumption because it is the only one that actually uses the GPS.

  - **Balanced Power Accuracy:** This mode has lower priority than High Accuracy mode. This mode only uses WPS, cell-ID and sensors to obtain location updates with a "block" level accuracy while keeping average power consumption very low.

  - **Low Power:** This mode has a lower priority than Balanced Power Accuracy mode. This mode only uses WPS, cell-ID and sensors to obtain location updates with a "city" level accuracy (10km) while keeping average power consumption extremely low.

  - **No Power:** This is the lowest priority mode. This mode does not use any location source to obtain location updates. Instead, obtains location updates requested by other third party applications in the Android platform. In this way the average power consumption is the lowest possible. The location interval update is unknown because it does not depend on our Location Client, as well as the accuracy, that could be anyone of the three previous options.

<table>
<thead>
<tr>
<th>PRIORITY</th>
<th>INTERVAL (s)</th>
<th>BATTERY DRAIN PER HOUR (%)</th>
<th>ACCURACY (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIGH_ACCURACY</td>
<td>5</td>
<td>7.25</td>
<td>10</td>
</tr>
<tr>
<td>BALANCED_POWER_ACCURACY</td>
<td>20</td>
<td>0.6</td>
<td>100</td>
</tr>
<tr>
<td>LOW_POWER</td>
<td>-</td>
<td>small</td>
<td>10.000</td>
</tr>
<tr>
<td>NO_POWER</td>
<td>N/A</td>
<td>small</td>
<td>variable</td>
</tr>
</tbody>
</table>

Table 3.2: Location Request priority modes

Table 3.2 shows a comparison of the power consumption per hour of each one of these priority modes.

### 3.2.4 Location Class

This is a data type representing a geographic location that can provide the following information:

- **Latitude & Longitude:** Returns the latitude and longitude of obtained location in degrees.

- **Timestamp:** Returns the UTC time of this fix, in milliseconds since January 1, 1970, that can be converted into a date, including day and time of when the location was obtained. This is not a monotonic value, which means it is not suitable to calculate time difference between location updates.
- **Elapsed Time:** Returns the time of this fix, in elapsed real-time since system boot. This time can be use to calculate time difference between two location samples because it actually is a monotonic value.

- **Accuracy:** Returns the estimated accuracy of the obtained location in meters.

- **Altitude:** Returns estimated altitude of the obtained location in meters. This is an information that can only be provided by GPS location source. The rest of location sources will not return any altitude value. Since we are not using GPS, we will discard this value.

- **Speed:** Returns estimated speed of the obtained location in meters. This is an information that can only be provided by GPS location source. The rest of location sources will not return any altitude value. Since we are not using GPS, we will discard this value.
4 Activity Recognition Components in MobilitApp

4.1 Activity Recognition Sources in Android devices

In opposition of what happens with location, there is basically one main source to detect user’s activity type: device sensors. Therefore, there is no real alternative to this source, and currently is the only possible option if we want to recognize activities. As we explained, there are four major sensors that are used to improve location updates, and although all of them can be somehow used for this purpose, in reality the main sensor that we are going to use is the accelerometer. Accelerometer has the lowest power consumption of all sensors, but is also the one that requires more processing of samples to provide accurate results.

4.2 Activity Recognition’s Tools

Android platform provide developers with multiple tools, including services, libraries, classes and methods to implement and develop their application. However, these tools are limited to obtain raw sensor data. Process raw sensor data in order to detect user’s activity with good accuracy is too complex to be designed and implemented to this project at the time of writing. Instead of that, we are going to use Google Activity Recognition API. These API provides a low power mechanism that can provide periodic updates of detected activity types by using low power sensors (basically accelerometer). We are going to process these samples in order to increase accuracy and efficiency. The most important classes that we are going to use from this API are presented below.

4.2.1 Activity Recognition Client

This is the main class from Activity Recognition API to detect user’s activity type data\[^{24}\]. Its principal functions are connect and disconnect to Google Services in background, and request activity recognition updates by starting an intent service.

4.2.2 Activity Recognition Result

This class\[^{25}\] returns a list with all probable activities. It also provides a method that directly returns the most probable activity along with its confidence level called getMostProbableActivity().

4.2.3 Detected Activity

This class\[^{26}\] returns an integer that indicates which activity type has bee detected by Activity Recognition API.
5 Data Format in MobilitApp

There are two major open data formats commonly used to store this type of data. On the one side we have XML[27], and on the other side we have JSON[28]. In order to choose one of them, we have to evaluate pros and cons of each type regarding a set of desirable features for our application. Among these features there are[29]:

- **Writing/Reading/Parsing Speed**: In general, JSON files have better speed performance than XML because their format is more simpler.

- **Simplicity**: JSON files are more simple than XML files.

- **Openness**: Both formats are open and completely free to use.

- **Interoperability**: Both formats are interoperable with multiple systems and languages.

- **Extensibility**: XML allows to store almost any kind of type of data while JSON is tied with classic format. However, this add more complexity to XML compared to JSON.

- **Human Readable**: Due its structure, JSON is more human readable than XML.

- **Sharing Data**: JSON is structure in arrays and objects which are easier and faster to transfer than XML trees.

In our application we need simplicity to have the best possible organization of data, interoperability to be able to interact with different systems and languages (like Java), we need to be able to transfer data fast in the event of future server functionalities, and most important of all we need good speed and performance when writing, reading and parsing data filed. Based on these aspects, we decided to store our mobility data using JSON format files.
Part III

MobilitApp Architecture
1 Main Activity

In our Main Activity class we are going to implement two main types of operations with different purposes:

- **Setup Operations:** It includes checking the availability of Google Play Services APK, checking the status of the location sources in the device, adding Google Map, and initiating location and activity recognition services.

- **Layout:** It includes organization of the application design, implementation and personalization of markers and lines on the map along with its functionality, setup of text views and connection with markers functionality, and implementation of location refresh and location history functions.

1.1 Setup Operations

1.1.1 Checking Google Play Services

Google Play services APK is essential to use Google Maps API[30], Fused Location Provider API and Activity Recognition API. It can be downloaded from the Android SDK Manager within the Android developing environment (Eclipse ADT). After downloaded it, we only have to import the package as a library into our project. Finally, we will have to implement the following block in the Android Manifest file to compile our application embedded with Google Play services:

```xml
<meta-data
  android:name="com.google.android.gms.version"
  android:value="@integer/google_play_services_version" />
```

In order to check availability of Google Play services, we will only need to implement a `checkGooglePlayServices()` function. This function will return SUCCESS if Google Play Services is installed and enabled to be used along with a toast (temporal little message for the user) to inform the user. In the opposite case, an error code indicating the reason why the connection could not be established will be returned and we will try to reconnect it. If the maximum timeout expires (CONNECTION_FAILURE_RESOLUTION_REQUEST) we will show an error dialog message.

This function is implemented in `onResume()` method to check Google Play services connection every time users resume the application in order to ensure the connection was not lost, or to reconnected in case it was.
private boolean checkGooglePlayServices()
{
    int resultCode = GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);
    if (ConnectionResult.SUCCESS == resultCode)
    {
        Toast.makeText(this, "Google Play Services Connected", Toast.LENGTH_LONG).show();
        return true;
    }
    else
    {
        Toast.makeText(this, "Google Play Services Error", Toast.LENGTH_LONG).show();
        Dialog dialog = GooglePlayServicesUtil.getErrorDialog(resultCode, this, AppUtils.CONNECTION_FAILURE_RESOLUTION_REQUEST);
        if (dialog != null)
        {
            ErrorDialogFragment errorFragment = new ErrorDialogFragment();
            errorFragment.setDialog(dialog);
            errorFragment.show(getFragmentManager(), AppUtils.ERROR_MESSAGE);
        }
        return false;
    }
}

1.1.2 Checking Location Sources

Once a connection with Google Play services has been established, we have to check that required location sources, according to our design criteria, are enabled in user’s device. We are going to proceed as the following diagram shows:

![Diagram showing the process of checking location sources]

Figure 1.1: Checking location sources diagram

1 Latest versions of Android have four possible location modes depending on which location sources are enabled in the device:

- **High Accuracy**: All location sources are enabled (GPS, Wi-Fi, and mobile networks).
- **Battery saving**: Only low power location sources are enabled (Wi-Fi and mobile networks).
- **Device only**: Only GPS is enabled.
- **Disabled**: All location sources are disabled.
2 If device is either in high accuracy or battery saving mode, the application is ready to start the location service in order to obtain location updates. Notice that our application only requires WPS and cell-ID active in order to obtain location updates, and therefore the status of the GPS does not affect our application because we do not use it.

3 If device is has either in device only or disabled mode, we have to redirect the user to location settings to change these modes into high accuracy or battery saving mode.

4 If the user decides not to enable one of these two mode, a message toast will be shown on the screen indicating that it is impossible for our application to obtain location data. The application will be checking location sources periodically if it is in foreground, because the function that does this task is implemented in onResume() method.

1.1.3 Adding Google Map

When we use Google Map API, we need to access to Google Servers from our application. In order to identify who are sending request to use the API, Google requires a to add unique key to our project. To generate this key we just need to provide our application’s package name and digital certificate, and provide it in the Google API Console. We also have to add the key in the Application Manifest in the following way:

```xml
<meta-data android:name="com.google.android.maps.v2.API_KEY" android:value="API_KEY"/>
```

Along with the Google Map API key, we have to add another block into the Manifest file. Since Google Maps uses OpenGL ES v2 to render the maps, we need to ensure that is actually installed in our application to successfully display the map on the screen.

```xml
<uses-feature android:glEsVersion="0x00020000" android:required="true"/>
```

Finally, we are going to add a function called setUpMap() that will add the map in our application. This function is implemented in onResume() method. In that way it will be part of the foreground loop of the application and can be updated or modified properly every time the user resumes the application.

```java
public void setUpMap(){
    //Getting reference to the MapFragment of activity_main.xml
    googleMap = ((MapFragment) getFragmentManager().findFragmentById(R.id.map)).getMap();

    //Enabling MyLocation Layer of Google Map
    googleMap.setMyLocationEnabled(true);

    //Enabling Compass Layer of Google Map
    googleMap.getUiSettings().setCompassEnabled(true);
}
```

1.1.4 Permissions

We have to add the following permissions in the Android Manifest file to allow our application to perform some required operations:
• **android.permission.INTERNET**: Allows application to have Internet access, like for example download Google Map.

• **com.google.android.gms.permission.ACTIVITY_RECOGNITION**: Allows application to use the Activity Recognition Client.

• **android.permission.ACCESS_NETWORK_STATE**: Allows application to check network status, which is useful for example to check if Google Play services are available.

• **android.permission.WRITE_EXTERNAL_STORAGE**: Allows application to save data in external storage unit of device to save our mobility data files.

• **android.permission.READ_EXTERNAL_STORAGE**: Allows application to read data from external storage unit of device to read our mobility data files.

• **com.google.android.providers.gsf.permission.READ_GSERVICES**: Allows application to access Google servers.

• **android.permission.ACCESS_COARSE_LOCATION**: Allows application to use Wi-Fi and mobile networks to obtain location samples.

### 1.1.5 Starting Application Services

#### Location Service

To start Location Service we have to create and intent with the service we want to start, and start it with the `startService()` command. If the service does not exist, it is created by calling its `onCreate()` method, but if it already exists, it is just started by calling its `onStartCommand()` method. This is the behavior we expect the service to have, as we determined when analyzing these types of services in the previous part.

```java
1 Intent location_intent = new Intent(this, LocationService.class);
2 this.startService(location_intent);
```

The most important part of this implementation is to decide from which application component start Location Service and where. We have done multiple test to see the real performance of our application when calling this service in different ways and from different methods.

- **Starting the Location Service from the Main Activity**: We observed that the service stopped running when the process of Main Activity was killed, even in the case where we created a new independent process for Location Service. We have determined that this situation happened because the service was somehow still bound with the Main Activity, and therefore the performance of the application did not reach our requirements. We also observed that Location Service was killed, on average, very fast by the system for being a background process (when sharing with Main Activity) or service process (when running own process).

- **Starting the Location Service from the Activity Recognition Service sharing same process**: We observed the same behavior that we experimented with the previous implementation.

- **Starting the Location Service from the Activity Recognition Service but running a new independent process**: We observed that this implementation was the one that work perfectly because both services, Location and Activity Recognition keep each other alive and safe from being killed by the system. Activity Recognition periodically calls the Location Service, which keeps it alive, and running in its own process that this time is truly
independent of Main Activity. Meanwhile, Location Service keeps Activity Recognition by periodically getting activity updates through a broadcast receiver, that as we explained, always works on foreground. In that way we not only ensure that Location Service is not killed but running with a service process instead of being force to use a foreground process, that consumes significantly more resources.

To create a service with its own running process we just need to add these lines into Android Manifest file:

```
<service
    android:name="com.example.pfc2014.LocationService"
    android:process=":LocationService">
</service>
```

**Activity Recognition Service**

To start the Activity Recognition Service we have to instantiate the Activity Recognition Client and connect it to the Google Play services.

```
activityRecognitionClient = new ActivityRecognitionClient(this, this, this);
activityRecognitionClient.connect();
```

We decided to add this functionality in the Main Activity component. As happened with Location Service, the implementation and performance of this service was not what we expected. Originally designed to be an intent service, we had to modify it because when the system killed Main Activity’s process, the service turned into a zombie service without a process to run its operations, and therefore the activity updates were stopped. We decided to turn this service into an a base service but with the behavior of an intent service, by deleting the `onHandleIntent()` and implement all the code in `onStartCommand()` method. In that way, in the event of the Main Activity process being killed, the Activity Recognition service is capable to recreate the activity process, perform its task, and kill the process again, as an intent service would do, but without the dependence on the Main Activity process.

```
@Override
public void onConnected(Bundle bundle) {
    if(activityRecognitionClient != null && activityRecognitionClient.isConnected()){
        Intent activity_intent = new Intent(this, ActivityRecognitionService.class);
        PendingIntent activityRecognitionIntent = PendingIntent.getService(this, 0, activity_intent,PendingIntent.FLAG_UPDATE_CURRENT);
        activityRecognitionClient.requestActivityUpdates(AppUtils.UPDATE_INTERVAL_ACTIVITY, activityRecognitionIntent);
    }
}
```

After Activity Recognition Client is successfully connected we have to setup the two important parameters of this service. The first one is `FLAG_UPDATE_CURRENT`, which determines that in the event that we want to call Activity Recognition Service but this already exists, the system will update it instead of trying to create it again. The second parameter is `UPDATE_INTERVAL_ACTIVITY`. According to our design criteria, we have to set this parameter to request activity updates every 5 seconds.
1.2 GUI Layout

In our application we want to show the citizen’s daily trips segmented according their activity type on a map along with tabs providing mobility information of each segment on the screen. To achieve that, we are going to do the following:

- Implement of application’s homepage design in the Main Activity.
- Graphical representation of user’s trips on the map.
- Display information of user’s trips on the map, and on tabs at the bottom of screen.

1.2.1 MobilitApp Homepage

The application homepage is setup one single time when the activity is created by importing the XML layout files. Our Main Activity design is divided in two parts with a proportion of 3 to 1. The top part, and the biggest one, includes the Google Map while the bottom part include tabs that will show user’s trip information.

![MobilitApp homepage layout](image)

Figure 1.2: MobilitApp homepage layout

1.2.2 Representing trips on the map

In this subsection we are going to explain how to represent user’s daily trips on the map. We need the information that we have parsed from JSON files that we created in order to draw personalize markers and lines that will provide a beautiful output based on our previous designs.

Markers

Markers are a graphical representation of a set of coordinates (latitude and longitude) on a map. Marker’s class is included in the Google Maps API. Adding a marker is very simple, because we only need to provide the location coordinates. We can easily personalize our markers.
by adding a title, a snippet, or we can even change the icons by default to provide a more beautiful
design for the user, like in the following example:

Depending on the activity type, we are going to use different icons for markers in order to
personalize design of trips that we have drawn on the map. For that reason we have designed a
set of icons to be used as markers in our application that are returned by a method we created
called `getActivityIcon()` depending on user’s activity type.

```java
private BitmapDescriptor getActivityIcon(String _activityType) {
    switch (_activityType) {
        case "on_foot":
            return BitmapDescriptorFactory.fromResource(R.drawable.ic_walk);
        case "vehicle":
            return BitmapDescriptorFactory.fromResource(R.drawable.ic_vehicle);
        case "bicycle":
            return BitmapDescriptorFactory.fromResource(R.drawable.ic_bicycle);
        case "still":
            return BitmapDescriptorFactory.fromResource(R.drawable.ic_place);
        case "transport":
            return BitmapDescriptorFactory.fromResource(R.drawable.ic_transport);
        case "unknown":
            return null;
    }
    return null;
}
```

Polylines

Polylines are lines used to connect markers on the map. Polyline’s class is included in the
Google Maps API. With polylines we can draw a list of consecutive coordinates, and personalize
the width and the color, among others things, like in the following example:

```java
googleMap.addPolyline(new PolylineOptions()
    .addAll(_latlng)
    .width(10)
    .color(getColor(_activity));
```

Depending on the activity type, we are going to use different colors for lines in order to
personalize design of trips that we have drawn on the map. For that reason we have picked a
palette of colors matching design of icons to be used in our application, that are returned by a
method we created called `getColor()` depending on user’s activity type.
private static int getColor(String activity) {
    switch (activity) {
        case "on_foot":
            return Color.rgb(255, 102, 51);
        case "vehicle":
            return Color.rgb(44, 62, 83);
        case "bicycle":
            return Color.rgb(205, 153, 102);
        case "transport":
            return Color.rgb(22, 189, 195);
        case "unknown":
            return Color.TRANSPARENT;
        case "still":
            return Color.TRANSPARENT;
    }
    return Color.TRANSPARENT;
}

Drawing trips function

Using markers and polylines, we have implemented a method that can draw citizen’s trip from mobility data obtained with our successive processing algorithms that provide an output as the one that can be seen in figure 1.3:

![Figure 1.3: Trip draw on the map using drawLocation() method](image)
```java
public void drawLocation (ArrayList<LatLng> latlng, String _first_time,
String _last_time, String _activity,
Double _distance, long _duration, Double _speed, int i) throws IOException{
    if(!latlng.isEmpty()){
        if (i == 0){
            googleMap.addMarker(new MarkerOptions()
                .position(latlng.get(0))
                .title(_activity)
                .snippet(_first_time + " − " + _last_time)
                .icon(getActivityIcon(_activity));
        } else{
            googleMap.addPolyline(new PolylineOptions()
                .addAll(latlng)
                .width(10)
                .color(getColor(_activity)));
            if (_activity.equals("still")){
                googleMap.addMarker(new MarkerOptions()
                    .position(latlng.get(latlng.size() - 1))
                    .title(_activity)
                    .snippet(_first_time + " − " + _last_time)
                    .icon(getActivityIcon(_activity));
            } else{
                CameraUpdate cameraUpdate = CameraUpdateFactory.newLatLngZoom(
                    latlng.get(latlng.size() - 5), 14);
                googleMap.animateCamera(cameraUpdate);
                googleMap.addMarker(new MarkerOptions()
                    .position(latlng.get(latlng.size() - 5))
                    .title(_activity)
                    .snippet(_first_time + " − " + _last_time)
                    .icon(getActivityIcon(_activity));
            }
        }
    } else{
        googleMap.addPolyline(new PolylineOptions()
            .addAll(latlng)
            .width(10)
            .color(getColor(_activity)));
        if (_activity.equals("still")){
            googleMap.addMarker(new MarkerOptions()
                .position(latlng.get(latlng.size() - 1))
                .title(_activity)
                .snippet(_first_time + " − " + _last_time)
                .icon(getActivityIcon(_activity));
        }
    }
}
```

1.2.3 Displaying trips information

In this subsection we are going to explain how to show mobility data of user’s on screen. First we have to do is implement in the Main Activity `onCreate()` method all TextViews we are going to use to display the information. At this point we will need one for duration, distance and average speed:

```java
distanceTextView = (TextView) findViewById(R.id.distance);
durationTextView = (TextView) findViewById(R.id.duration);
speedTextView = (TextView) findViewById(R.id.speed);
```

To display the text we just simply need to use `setText()`, included in TextView class, with each one of the fields:

The next step is obtain mobility data from stored JSON files. The way these files are written and parsed is explained in another chapter. Basically we have to obtain the information about activity, distance, duration and average speed, and add to an ArrayList that we are going to use to set these information in the TextView tabs:
Finally we are going add a new functionality to our markers. The objective is that when user clicks on the title label of each one of the added markers regarding each one of the obtained segments, the information of total distance, total duration and average speed will be shown on TextViews. If the user click another marker, then current information will be replaced with new information from selected segment. To do that we have to set a click listener that will read all segments and select information of the specific segment selected by the user. An example of this functionality can be seen in figure 1.4.

```java
distanceTextView.setText("TOTAL DISTANCE " + distance + " Kms");
durationTextView.setText(
  "TOTAL DURATION " + Long.toString(hours) + " h " +
  Long.toString(minutes) + " min " +
  Long.toString(seconds) + " s ");
speedTextView.setText("AVERAGE SPEED " + speed + " Km/h");
```

```java
marker = new Segment();
marker.setActivity("m"+Integer.toString(i));
marker.setTotalDistance(distance);
marker.setTotalDuration(duration);
marker.setTotalAverageSpeed(speed);
markers.add(i,marker);
```

```java
googleMap.setOnInfoWindowClickListener(
  new OnInfoWindowClickListener()
  {
    @Override
    public void onInfoWindowClick(Marker marker)
    {
      int i=0;
      match = false;
      while(!match)
      {
        if (marker.getId().equals(markers.get(i).getActivity()))
          match = true;
        else i++;
      }
      long seconds = markers.get(i).getTotalDuration();
      long hours = seconds/(60∗60);
      seconds = seconds % (60∗60);
      long minutes = seconds/60;
      seconds = seconds % 60;
      String distance = String.format("%.2f", (markers.get(i).getTotalDistance()/1000));
      String speed = String.format("%.2f", markers.get(i).getAverageSpeed());
      distanceTextView.setText(
        "TOTAL DISTANCE " + distance + " Kms");
durationTextView.setText(
  "TOTAL DURATION " + Long.toString(hours) + " h " +
  Long.toString(minutes) + " min " +
  Long.toString(seconds) + " s ");
speedTextView.setText(
  "AVERAGE SPEED " + speed + " Km/h");
    }
  });
```
Figure 1.4: MobilitApp Layout
2 Activity Recognition Service

Architecture

Activity Recognition service objective is detect user’s activity and send the result every 5 seconds back to the Location Service for further processing. This service do not need to create its own process to work because it uses Main Activity process. By using a pending intent we allow to Activity Recognition service the possibility to response to periodic request for activity updates. This results in the following behavior. If the Main Activity process is killed, the bound between this service and the process it is not, and therefore the service can temporarily recreate the process, perform its task, and let the process to die again in a cyclic basis. The architecture of service that we have implemented is the following:

```java
public class ActivityRecognitionService extends Service{
    static int activityType;
    String activityName;

    @Override
    public int onStartCommand(final Intent intent, int flags, int startId) {

        //If the incoming intent contains an update
        if(ActivityRecognitionResult.hasResult(intent)) {

            // Get the update
            ActivityRecognitionResult result = ActivityRecognitionResult.extractResult(intent);

            // Get the most probable activity
            DetectedActivity mostProbableActivity = result.getMostProbableActivity();

            //Get an integer describing the type of activity
            activityType = mostProbableActivity.getType();
            activityName = getNameFromType(activityType);

            //Retrieved activity is sent to LocationServices
            Intent intentActivity = new Intent("activityRecognition");
            sendActivityLocationBroadcast(intentActivity, activityName);

            return START_STICKY;
        }
    }
}
```

The functioning of Activity Recognition service is very simple. When it receives an intent request for an activity update, it uses Activity Recognition Result class to extract and obtain list with all possible detected activities, from which we get the most probable one.

Activity Detection

Since the activity type is in integer format, we use a switch/case function called `getNameFromType()` that returns a string with the name of the detected activity. The types of activity we are going to detect are:

- **on_foot**: Activity type returned if the citizen is either walking or running
- **bicycle**: Activity type returned if the citizen is on a bicycle
- **vehicle**: activity type returned if the citizen is on a motor vehicle (car, motorbike or bus)
- **still**: Activity type returned if the citizen is not moving
- **unknown**: Activity type returned if Activity Recognition API is not capable to estimate the actual activity
- **transport**: Activity type created by us to be used on segment processing algorithm that will be returned if the citizen is on railway transport (metro, tram or train)

```java
private String getNameFromType(int activityType) {
    switch (activityType) {
        case DetectedActivity.IN_VEHICLE:
            return "vehicle";
        case DetectedActivity.ON_BICYCLE:
            return "bicycle";
        case DetectedActivity.ON_FOOT:
            return "on_foot";
        case DetectedActivity.STILL:
            return "still";
        case DetectedActivity.UNKNOWN:
            return "unknown";
        case DetectedActivity.TILTING:
            //tilting changed to still
            return "still";
    }
    return "null";
}
```

**Communication**

Finally, to conclude with Activity Recognition service operation, we create a broadcast receiver to send detected activity type to Location Service.

```java
private void sendActivityLocationBroadcast(Intent _intent, String _activity) {
    _intent.putExtra("activity", _activity);
    sendBroadcast(_intent);
}
```
3 Location Service

In this chapter we are going to take a greater depth on Location Service architecture in order to explain how it is finally designed, which methods and functions are implemented, and which is the expected performance.

3.1 Starting Location Service

After calling and creating the Location Service from Activity Recognition Service, we have to actually create the Location Client that will connect with Google Play Services and start requesting updates by implementing the following block of code in `onCreate()` method of the service:

```java
locationClient = new LocationClient (this, this, this);
locationClient.connect();
```

Once connection has been established, we have to set up the main parameters, with Location Request class, before starting requesting location updates. According to our design criteria, we are going to set the priority mode to use only WPS, Cell-ID and sensors to obtain location with a block level accuracy, ensure that our interval update is no lower than 20 seconds, and at the same time try to keep interval not too much higher than 20 seconds. The following code shows how to setup all these parameters:

```java
@Override
public void onConnected(Bundle bundle) {
    if (locationClient != null && locationClient.isConnected() && !locationConnected) {
        locationConnected = true;
        locationRequest = LocationRequest.create()
            .setPriority(LocationRequest.PRIORITY_BALANCED_POWER_ACCURACY)
            .setInterval(AppUtils.UPDATE_INTERVAL_LOCATION)
            .setFastestInterval(AppUtils.FASTEST_INTERVAL_LOCATION);
        locationClient.requestLocationUpdates(locationRequest, this);
    } else locationClient.removeLocationUpdates(this);
}
```

3.2 Activity Processing Algorithm

Activity Processing algorithm has three nested stages in order to provide user’s activity type with a high confidence level. The first stage is a broadcast receiver that obtains the activity updates sent by the Activity Recognition Service in intervals of 5 seconds.

Activity Detection Receiver

```java
private BroadcastReceiver activityReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        activityName = activityProcessing (intent.getStringExtra("activity"));
    }
};
```
Activity Processing Algorithm

The second stage is implemented inside the receiver. We implement a function that stores consecutive activity updates during the segment interval time (2 minutes), which means a total of 24 activity samples, in an ArrayList called bufferActivityArrayList. Once we have all the samples, we will count how many samples of each activity type we have obtained.

```java
private String activityProcessing (String _activity){
    vehicle_count=0; bicycle_count=0; on_foot_count=0; still_count=0; unknown_count=0;
    if(location_on){
        //Get and store activity updates until ACTIVITY_SAMPLES
        if (i<AppUtils.ACTIVITY_SAMPLES)
            //Get and store a string describing the type of activity
            bufferActivityArrayList.add(i,_activity);
            i++;
    } //When we got all the samples required, we proceed to the processing of data
    else {
        //Reset counter for new samples
        i=0;
        //iterate the array to obtain all the activities and respective confidences
        for(int j=0;j<AppUtils.ACTIVITY_SAMPLES;j++){
            switch(bufferActivityArrayList.get(j)) {
                case "still":
                    still_count++;
                    break;
                case "vehicle":
                    vehicle_count++;
                    break;
                case "bicycle":
                    bicycle_count++;
                    break;
                case "on_foot":
                    on_foot_count++;
                    break;
                case "unknown":
                    unknown_count++;
                    break;
                default: break;
            }
        } //Estimation of the activity
        activityName = activityEstimation(
            on_foot_count, bicycle_count, still_count, vehicle_count, unknown_count);
        //Delete all detected activities waiting for the new activity samples
        bufferActivityArrayList.clear();
    }
    return activityName;
}
```

Activity Decision Tree

The third stage is a decision tree that will return the most probable activity. To return the activity type we are going to use a global variable that will be used in segment processing algorithm. In case of two or more activities have the same confidence level, we have established a priority order that goes: on foot, vehicle, bicycle, unknown and still.
```java
private String activityEstimation(
    int isOnfoot, int isBicycle, int isStill, int isVehicle, int isUnknown)
{
    // on_foot is the most probable activity
    if ((isOnfoot - isBicycle) > 0 && (isOnfoot - isStill) > 0 &&
        (isOnfoot - isVehicle) > 0 && (isOnfoot - isUnknown) > 0)
    return "on_foot";

    // bicycle is the most probable activity
    else if ((isBicycle - isOnfoot) > 0 && (isBicycle - isStill) > 0 &&
        (isBicycle - isVehicle) > 0 && (isBicycle - isUnknown) > 0)
    return "bicycle";

    // still is the most probable activity
    else if ((isStill - isOnfoot) > 0 && (isStill - isBicycle) > 0 &&
        (isStill - isVehicle) > 0 && (isStill - isUnknown) > 0)
    return "still";

    // vehicle is the most probable activity
    else if ((isVehicle - isOnfoot) > 0 && (isVehicle - isBicycle) > 0 &&
        (isVehicle - isStill) > 0 && (isVehicle - isUnknown) > 0)
    return "vehicle";

    // unknown is the most probable activity
    else if ((isUnknown - isOnfoot) > 0 && (isUnknown - isBicycle) > 0 &&
        (isUnknown - isStill) > 0 && (isUnknown - isVehicle) > 0)
    return "unknown";

    // in case of equal probability of activity, on_foot has priority over the rest
    else if ((isOnfoot == isBicycle) || (isOnfoot == isStill) ||
        (isOnfoot == isVehicle) || (isOnfoot == isUnknown))
    return "on_foot";

    // in case of equal probability of activity, after on_foot, vehicle has priority over the rest
    else if ((isVehicle == isBicycle) || (isVehicle == isStill) ||
        (isVehicle == isUnknown))
    return "vehicle";

    // in case of equal probability of activity, bicycle has priority over still and unknown
    else if ((isBicycle == isStill) || (isBicycle == isUnknown))
    return "bicycle";

    // in case of equal probability of activity, unknown has priority over still
    else if ((isStill == isUnknown))
    return "unknown";

    else return "still";
}
```

### 3.3 Location Processing Algorithm

**Obtaining Location Updates**

Location Processing algorithm has three consecutive stages. In the first stage, we use a Location Listener to obtain location updates every 20 seconds approximately. We get the location samples in `onLocationChanged()` method as long as they are not null. We have created a global ArrayList called `tempLocationArrayList` to store the location samples. We are going to use this same ArrayList to process location samples in the next processing algorithm. At the same time we are periodically calling the broadcast receiver that contains the activity algorithm process. Once we have a detected activity and reach the minimum segment time of 2 minutes, we proceed to the next stage.
3.3.1 Calculation of Mobility Data Parameters

In the second stage we use samples location stored in the ArrayList to obtain the total distance, total time and average speed of the user. First, we calculate the distance and time deltas between consecutive locations. Then we add all of them to obtain the total value of distance and time. With delta distance and delta times we calculate the speed of each delta segment in order to calculate the average speed during the whole segment time.

```
//Distance, time and average speed calculation
for (int j=0; j<size-1;j++){
    //Distance between consecutive locations, in meters
    segment_delta_distance = AppUtils.distanceCalculation(
        tempLocationArrayList.get(j).getLatitude(),
        tempLocationArrayList.get(j).getLongitude(),
        tempLocationArrayList.get(j+1).getLatitude(),
        tempLocationArrayList.get(j+1).getLongitude());
    //Sum of all distance deltas of the segment
    total_segment_delta_distance =
        total_segment_delta_distance + segment_delta_distance;
    //Time between consecutive locations, in seconds
    segment_delta_time =
        (tempLocationArrayList.get(j+1).getElapsedRealtimeNanos() - tempLocationArrayList.get(j).getElapsedRealtimeNanos()) / AppUtils.NANOSECONDS_PER_SECOND;
    //Sum of all distance deltas of the segment
    total_segment_delta_time =
        total_segment_delta_time + segment_delta_time;
    //Sum of all speed deltas of the segment, in Km/h
    total_segment_delta_speed =
        total_segment_delta_speed + ((segment_delta_distance)*3.6)/segment_delta_time;
}
```

3.3.2 Creating Mobility Data Segments

In the third stage we are going to create the mobility data segments, including location points, activity type, distance, duration, average speed, and date of first and last location point. Segment is a class that we have created to store mobility data in a way that can be easily processed, stored and parsed. To create the segment class we have to define the private variable, build the constructors and implement setting and getting methods for each one of types of data in order
to store and read the mobility data. Segment class format can be seen at annexes. In order to process the segment, we have to create them first. The procedure to do it is always the same. We get the size of location ArrayList in order to know how many location samples we have. Then we create a new empty segment and start setting all its fields with mobility data. First location field is set with the last location from the previous segment to keep the linearity in the time of segment, while last location field is set with last location from current segment. After we set all the field of the segment, we add it into a global ArrayList, where all obtained segments are going to be stored. Then we save last location from this segment to add it as first location of the new one. Finally, we clear location ArrayList in order to store the new location updates from the next segment.

```java
int segment_size = tempLocationArrayList.size();

//create new segment to store the new values
Segment segment = new Segment();

//last location from previous segment stored previously
segment.setFirstLocation(tempLastLocation);
segment.setLastLocation(tempLocationArrayList.get(segment_size - 1));
segment.setTotalDistance(total_segment_delta_distance);
segment.setTotalDuration(segment_total_duration);
segment.setTotalAverageSpeed(average_segment_speed);
segment.setActivity("vehicle");
segment.updateLocationPoints(tempLocationArrayList);

//add current segment in the next spot
segmentArrayList.add(segment);

//store last location
tempLastLocation = tempLocationArrayList.get(segment_size - 1);

//clear location ArrayList
tempLocationArrayList.clear();
```

However, not all activity types following this procedure when creating a new segment. Activity type "still" has slightly different way because on one side, we actually need one single location, and on the other side, parameters as distance or average speed are useless. For that reason, we will select the location sample from the location ArrayList with best accuracy and store it in the segment, instead of storing all location points. As for distance and average speed, we will store no value for them because "still" means being in the same place. The rest works in the same way as explained before for the rest of activity types.
```java
int segment_size = tempLocationArrayList.size();
// create new segment to store data
segment = new Segment();

if(activityName.equals("still")){
    if (segment_size > 1){
        tempPlace = tempLocationArrayList.get(0);
        for (int j=0; j< segment_size;j++){
            if(tempPlace.getAccuracy() < tempLocationArrayList.get(j).getAccuracy())
                tempPlace = tempLocationArrayList.get(j);
        }
    }
    // add parameters to created segment
    segment.setFirstLocation(tempLocationArrayList.get(0));
    segment.setLastLocation(tempLocationArrayList.get(segment_size-1));
    segment.setTotalDistance(0);
    segment.setTotalDuration(temp_duration);
    segment.setTotalAverageSpeed(0);
    segment.setActivity("still");
    segment.addSingleLocationPoint(tempPlace);

    // store still place as Last Location (with the modified time)
    tempLastLocation = tempLocationArrayList.get(segment_size-1);

    // store segment in the last position of the segment array (in this case is the first one)
    segmentArrayList.add(segment);

    tempLocationArrayList.clear();
}
```

### 3.4 Segments Processing Algorithms

#### 3.4.1 Assumptions

Before processing segments obtained from previous algorithms, we have to establish several assumptions regarding different activity types in order to be able to estimate them with the maximum precision. At this point of time, distance and average speed are the parameters that we are going to increase efficiency and accuracy of segments estimation. Based on field tests, several references\[36]\[37]\[38\] about average transport speeds in the metropolitan region of Barcelona, we have determined the following assumptions:

- Citizen’s are going to be considered in a place, i.e., that they are not moving from one location, if the maximum displacement between any pair of obtained location samples is lower than a "block" distance. In this case we set block distance to be 75 meters if the returned activity is "still". In the same direction, we are going to consider that the average speed has to be lower than 1 Km/h.

- We are going to consider that citizens are moving if the displacement between two consecutive location is higher than 25 meters, except when we activity type "still" applies.

- We are going to consider than the maximum speed for a citizen that is either walking or running is 9 Km/h. Our objective is to track mobility patterns in a daily basis, and therefor we consider that running patterns will not be often or during long periods of time.

- We are going to consider that the maximum speed for a citizen that is on a bicycle is 15 Km/h.
• We are going to consider that the minimum speed that a motor vehicle can have in the metropolitan region is 10 Km/h.

• We are going to consider that during the segment interval time, the minimum distance that a railway transport must cover is 450 meters, based on the average distance between stations and the theoretical average speed of metro, tram and train.

3.4.2 Implementation

Using the segments created with the previous algorithms we are going process the segment data, in order to correct possible activity recognition and location errors increasing that way the accuracy and efficiency of segment estimation, that will have a direct impact on the estimation of citizens trips.

The actual algorithm is a switch/case function nested with if/else conditions. Depending on the activity we have detected, the activity detected from previous segments, and the different assumption for each type of activity we decide which is the correct estimation of the segment. If the new segment has detected the same activity type of the previous segment, then we just update segment fields with new data and update segment ArrayList. On the contrary, we add the new segment in the ArrayList with its data values

\[1\]

\[1\]The full code of segment processing algorithm can be found in https://sertel.upc.edu/ maguilar/indexold.html
4 Segment Files

In this chapter we are going to explain how to write, parse and store a JSON file containing mobility data from processed segments. With every segment that we have processed, we update the JSON file that contains mobility data of the current day. We write a JSON file on devices external storage. In that case, in the event of any kind of error, crash, etc that may occur to our application, we will ensure that we will be able to read the latest update of user’s mobility data available at the time MobilitApp was shut down.

4.1 Writing a JSON file

Writing a JSON file in Android has to two different aspects that we have to consider when implementing to code. One aspect is how actually write a file in Android, and the other is how to structure our mobility data in the JSON file.

```java
public static void writeSegment (String filename, ArrayList<Segment> segmentList){
    String segment_json = null;
    try{
        //create the segments array
        JSONArray segments = new JSONArray();
        for (int k=0; k < segmentList.size();k++){
            //create the objects that will be in each spot of the segment array
            JSONObject single_segment = new JSONObject();
            single_segment.put("activity", _segmentList.get(k).getActivity());
            single_segment.put("distance (m)", _segmentList.get(k).getTotalDistance());
            single_segment.put("duration (s)", _segmentList.get(k).getTotalDuration());
            single_segment.put("speed (Km/h)", _segmentList.get(k).getAverageSpeed());
            single_segment.put("first time", AppUtils.getTime(_segmentList.get(k).getFirstLocation().getTime()));
            single_segment.put("last time", AppUtils.getTime(_segmentList.get(k).getLastLocation().getTime()));
            //create an array inside the object for each spot of the segment array
            JSONArray location_points = new JSONArray();
            for(int j=0; j < _segmentList.get(k).getLocationPoints().size();j++){
                location_points.put(_segmentList.get(k).getLocationPoints().get(j).getLatitude());
                location_points.put(_segmentList.get(k).getLocationPoints().get(j).getLongitude());
                location_points.put(AppUtils.getTime(_segmentList.get(k).getLocationPoints().get(j).getTime()));
            }
            single_segment.put("location", location_points);
        segments.put(k, single_segment);
    }
    JSONObject json = new JSONObject();
    json.put("segments", segments);
    System.out.print(segments);
    }catch (JSONException e) e.printStackTrace();
}
```

In JSON, we can use two different classes, JSONObject and JSONArray. According to [28], a JSONObject is an unordered collection of name/value pairs while a JSONArray is an ordered sequence of values. In both cases, the values that can be used are Boolean, JSONArray,
JSONObject, Number (integer, long, double,...) or String. This means that we can have arrays inside objects, objects inside arrays, and almost any kind of nested combination using these data structures.

Our JSON files will be written following the structure of segment class we designed. We will have to go through all the ArrayList segments and put field by field all information of each segment into a JSONObject. The location field contains an array of multiple location samples with format of latitude, longitude, and time. We will have to create a JSONArray that will contain all these values, and then put this array inside JSONObject containing data of same segment. Finally, we will put all JSONObjects, each one of them with mobility data from different segments, into a JSONArray. This array will contain all the obtained segments, and at the same time it will facilitate its further parsing and processing.

```java
public static String getTime(long timestamp){
    Calendar cal = Calendar.getInstance(Locale.ENGLISH);
    cal.setTimelnMillis(timestamp);
    String day = DateFormat.format("HH:mm:ss", cal).toString();
    return day;
}
```

Since we cannot write Location data type into a JSON file, we will only write the elements that contain the information we want to save. Regarding location field, we are going to obtain both coordinates (latitude and longitude) from location samples along with the timestamp of each sample. However, we have to convert the timestamp into a string with the format that shows hours, minutes and seconds. To accomplish that we will implement a simple function `getTime()` that will return any timestamp into a string with desired format. For the first location and last location fields, we are going to save their timestamps, also converted into time string format.

Back to the first aspect we mentioned, to write a file in Android we just need to use a `FileWriter` structure, that will continuously overwrite JSON file with newer segment data in string format. The reason to overwrite the data instead of just appending is the format of the JSON file. Appending JSONObjects and JSONArrays significantly increases the complexity of the writing process because we would have to restructure every time the file format, if we want to periodically save our mobility data.

```java
try{
    FileWriter file = new FileWriter(_filename, false);
    file.append(json.toString(1));
    file.flush();
    file.close();
}
```catch (IOException e) e.printStackTrace();

Finally, we are going to store JSON files in external storage of Android device. The format name of the files will be "yyyyMMddlocation_segment.json" in order to achieve an easy organization of daily trips per year, month and day.

```java
writeSegment("extSdCard/" + AppUtils.getDay(segmentArrayList.get(0).getFirstLocation().getTime()) + "location_segment.json", segmentArrayList);
```

### 4.2 Parsing a JSON file

Parsing is the analysis process of a syntactical structure (string or text) written in computer language, according to established set of rules[39]. In our application, want to parse JSON file with daily mobility data from citizens, that we have created, in order to represent these data on a
map. The attributes we have to extract are formatted according the writing structure we decided to use. The following extract of a real JSON file shows how it is structured, a general JSONArray containing all segments, each segment wrapped into a JSONObject, and location points putted inside JSONObject in the form of a JSONArray.

```json
{
    "segments": [
        {
            "duration (s)": 159,
            "speed (Km/h)": 4.123780688321875,
            "last time": "2014−06−16 11:25:30",
            "first time": "2014−06−16 11:22:49",
            "location": [
                41.3889738,
                2.1140225,
                "11:22:49",
                41.3889738,
                2.1140225,
                "11:23:09",
                41.3888648,
                2.1124391,
                "11:23:29",
                41.3888648,
                2.1124391,
                "11:23:49",
                41.3887638,
                2.1120605,
                "11:24:13",
                41.3887638,
                2.1120605,
                "11:24:40",
                41.3887012,
                2.1121054,
                "11:25:07",
                41.3886364,
                2.1119183,
                "11:25:30"
            ],
            "distance (m)": 191.26366849548702,
            "activity": "on_foot"
        }
    ]
}
```
Parsing a JSON file has two main aspects that we need to consider, as happens with the writing process. The first aspect is how to read a file in Android, and the second aspect is how to do the parsing. To read a file from Android external storage, we only need to name and path of the file, and use an implementation like the following to obtain a string with all data in the file:

```java
try {
    File readFile = new File(filename);
    FileInputStream stream = new FileInputStream(readFile);
    String jsonStr = null;
    try {
        FileChannel fc = stream.getChannel();
        MappedByteBuffer bb = fc.map(FileChannel.MapMode.READ_ONLY, 0, fc.size());
        jsonStr = Charset.defaultCharset().decode(bb).toString();
    } finally {
        stream.close();
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

The second aspect is parsing of the file. Basically we have to proceed in the opposite direction of writing the file. We will have to deconstruct JSONArrays and JSONObjects. To do the parsing we will need to create a variable for each data field and an ArrayList for location field.

```java
public void readSegment (String filename) {
    String _first_time, _last_time, _activity; Double _distance, _speed; long _duration;
    ArrayList<LatLng> _latlngList = null;
    JSONObject jsonObj = new JSONObject(jsonStr);
    // Getting JSON Array node
    segmentsData = jsonObj.getJSONArray("segments");
    for (int i = 0; i < segmentsData.length(); i++) {
        JSONObject segment = segmentsData.getJSONObject(i);
        _first_time = segment.getString("first time");
        _last_time = segment.getString("last time");
        _activity = segment.getString("activity");
        _distance = segment.getDouble("distance (m)");
        _duration = segment.getLong("duration (s)");
        _speed = segment.getDouble("speed (Km/h)");
        _latlngList = new ArrayList<LatLng>();
        if (_latlng != null) _latlngList.add(_latlng);
        JSONArray location_points = segment.getJSONArray("location");
        for (int j=0; j < location_points.length(); j++){
            LatLng _latlng = new LatLng(location_points.optDouble(j),location_points.optDouble(j+1));
            _latlngList.add(_latlng);
            j = j+3;
        }
    }
    catch (JSONException e) e.printStackTrace();
}
```
However, instead of storing location data into Location class format, we will use a format that only contains latitude and longitude, *LatLng*. We are doing this because markers, polylines and other Google Map classes only uses these format of data to draw locations. Therefore is more efficient convert data only one time into the most adequate format to achieve our purposes than do it twice. Besides that, parsing procedure is very straightforward, we get all JSONObjects by going through the JSONArray that contains all segments, and we extract each field to save it in private variables. We do the same for the JSONArray that contains all location samples. Once we parsed all data from JSON file, we can use our `drawLocation()` method to draw it on *MobilitApp* map:

```
1 drawLocation(latlngList, first_time, last_time, activity, distance, duration, speed, i);
```
5 MobilitApp Summary

Figure 5.1: Diagram of MobilitApp
Part IV

Results & Future Development
1 MobilitApp Achieved Goals

In this chapter we are going to present all the results we have obtained from multiple tests on our *MobilitApp* application\(^1\).

In our *MobilitApp* application we are capable to estimate citizen trips of each day. Each trip is a combination from consecutive segments including information of location points, detected activity and other mobility data for each one. *MobilitApp* can store the mobility data in JSON files, that can be retrieved by user’s to see current daily trip as well as previous trips. By clicking on the segments that are part of each trip, user can also see distance, duration and average speed of selected segment.

![Figure 1.1: Screenshot of results obtained with MobilitApp](image)

**Detectable Activities**

*MobilitApp* is capable to detect the following citizen’s activity:

- **Transport:**
  - Motor (car or bus) - called "*vehicle*" in the app.
  - Railway (metro, tram or train) - called *transport* in the app.

- On foot (walk or run)

\(^1\)The source code is available at [https://sertel.upc.edu/maguilar/indexold.html](https://sertel.upc.edu/maguilar/indexold.html)
- Bicycle
- Place (stationary)

However, we are not capable to meet our initial requirements of 75% detecting in some activities or in some scenarios. While "on foot", "bicycle", "still" and "vehicle" estimation provides good results with required accuracy, when detecting railway transport, the application results have shown some inconsistency. For example, depending if the user was sitting or standing, the activity type detection in the metro may change. Some tests on the metro returned "vehicle" as activity type detected, and few ones estimate that the user was walking.

Figure 1.2: Detectable Activity Types by MobilitApp

Location Estimation

*MobilitApp* can obtain citizens location in the city of Barcelona and its metropolitan region using only WPS, cell-ID and device sensors. We can consistently track citizens outdoor, indoor and underground achieving a block level accuracy and maximum coverage from our requirements. However, we have identified four major issues from our application performance:

- Based on the way we designed the application, when no signal from location sources is available *MobilitApp* transition to a standby state where neither location or activity type updates are received. We loss citizens tracking.

- In the city, when our application uses only cell-ID and sensors as location sources (WPS is not available), we have identified that changing from one mobile network cell to another can provide location estimation with accuracy lower than block level (more than 150 meters), which at the same time creates a trip with "strange" mobility patterns that do not match citizen real behavior.

- In some locations we have experimented some issues regarding our assumption of a block radius of 75 meters. In most of apartments and buildings of the metropolitan region we obtain desired output from this assumption because our application do not identify little movement within places. However, in certain places with a larger radius, like UPC - Campus Nord, our application identifies such movements in these areas, which is in fact a real behavior, but is data that have no significant value for our purposes.

- We have identified some major differences in location estimation depending on metro lines that citizens are using. For example, in several tests in the city of Barcelona, we have achieved both good accuracy and good consistency in location data when we were using metro line 1. On the contrary, location data from L3 resulted in significantly lower both in accuracy and consistency.

Obtainable Information

*MobilitApp* can provide start/end time of each one of the segments, its distance, duration and average speed with an optimal accuracy. No major issues have been identified with the process to get this information, or its quality.
Figure 1.3: Screenshot of results obtained with MobilitApp

**Average Power Consumption**

After multiple tests on our application performance, we can say that we have achieved the average power consumption objectives that we set on our design criteria. We have identified the following average power consumption parameters after multiple test with our application:

- Approximately 0.5% per hour when the main location source is WPS.
- Approximately 1.0% per hour when the main location source is cell-ID.
- Detected peaks of 2% per hours in scenarios with very low signal (both WPS and cell-ID).
2 MobilitApp Future Implementations

In this chapter, we are presenting all future upgrades that we will implement in our application that could not have been possible because we had a limited time to present this current implementation. These future implementations are the following:

**Average Power Consumption:**
We can improve the efficiency and reduce even more the average power consumption of MobilitApp although we met the requirements we set. To achieve that we could set a variable interval time for location and activity type updates depending on the current citizen activity. We also could decrease listening time for new location updates, i.e., only listen for location updates when there is actual some data to obtain. We could also use no power mode of location client to obtain location from 3rd-party application.

**Activity Processing Algorithms:**
We can improve activity detection algorithm by processing raw sensor data instead of using Google Activity Recognition API. That was impossible for us to do in such a short time cause it requires to use of new processing trees, processing techniques and thresholds to detect full set of activities that can be detected in metropolitan region like it is done in [41][42][43].

**Server Implementation:**
With more time, we could have implemented a server, in order to send there all JSON files. In that way anyone with granted access from project members would be able to download these mobility data for analysis purposes.

**CO₂ Estimation:**
We can add equations to calculate grams of CO₂ consumed by each activity type. Each type of vehicle and transport have a different average consumption based on multiple features, which require a level of complexity that we could not achieve with our given time.

**Profile Implementation:**
We can create user profiles to obtain demographic data from citizen, like age or gender when they login in our application. With these data we could create new fare integration tickets for groups of population with similar mobility patterns and needs.

**News & Real Time Updates on Urban Life & Events:**
We can add functionalities that provide services based on citizen’s mobility pattern. For example, if the citizen is close to a metro or bus station that he or she uses regularly, based on data we have, we could send a notification in the event of one or several station in the transport line are out of service, an accident happens, or any kind of incident that could affect citizen’s displacement within the metropolitan region.
Part V

Bibliography
Bibliography


[34] Android Developers, *GoogleMap.OnInfoWindowClickListener*,


